Hypertesting: The Case for Automated Testing of Hyperproperties

Johannes Kinder

Department of Computer Science
Royal Holloway, University of London
johannes.kinder@rhul.ac.uk

Abstract. Proof systems give absolute guarantees but are notoriously difficult to use for non-experts. Bug-finding tools make no completeness guarantees but offer a high degree of automation and are relatively easy to use for developers. For safety properties, the effectiveness of automatic test generation and bug finding is well established. For security properties like non-interference, which cannot be expressed as properties of a single program execution (i.e., hyperproperties), methods for testing and bug finding are in their infancy. In general, violations of hyperproperties cannot be expressed with a single test case like safety properties, so existing bug finding methods do not apply.
This paper takes the position that we should fill this gap in the arsenal of verification technology and outlines concepts and tools for the next generation of bug finding systems. In particular, we aim to establish a generalized concept for the generation of "hypertests", sets of tests that either provide some level of confidence in the system or give counterexamples to hyperproperties. As concrete instances of hypertesting, we foresee automated testing for violations of secure information flow and of numeric and cryptographic properties of programs.

1 Introduction

Numerous incidents have shown that many of the software systems we use are vulnerable to malicious attacks. Consequently, efforts are under way to secure especially critical parts of the infrastructure, such as commonly used libraries and protocols. For instance, the highly publicized "Heartbleed" vulnerability has led to the launch of an extensive code audit and rewriting campaign for the OpenSSL library[1]. Specifications or small reference implementations of protocols can be proven to adhere to certain security properties. These methods are highly labor-intensive and are thus inherently limited to few, especially valuable components. Impressive recent examples include the seL4 verified operating system kernel (20 person years for 10 KLOC) [18] or the verified TLS reference implementation [5].

Application software, however, is not usually regarded as security critical, and end users tend to select their favorite software based on features instead of security. Consequently, the focus of developers working against tight deadlines is on adding features

[1] http://www.libressl.org

instead of spending precious time on verifying security. Still, users entrust such applications with an increasing amount of sensitive information, as they use smartphones and web applications to manage their finances, health records, and personal contacts.

A major part of the problem is due to software bugs: they can range from simple syntactic typos or copy/paste mistakes (e.g., Apple's infamous "goto fail" bug [13]) to design errors or wrong usage of library interfaces. Traditionally, low-level bugs like buffer overflows have received most attention due to being easily exploited. But as modern programming languages and bug finding tools rule out increasingly more of these low-hanging fruits, sophisticated government- or organized crime-sponsored attackers will turn to other classes of weaknesses, such as information leaks or cryptographic vulnerabilities [7, 26, 20, 27].

Bug finding tools have helped bring verification technology "to the masses". Instead of trying to prove a (safety) property, they focus on finding counterexamples (bugs). This is substantially more robust and less prone to the false positives dreaded by software developers. No similar machinery is available to developers trying to harden their code against violations of more complex properties. Much progress has been made in using proof systems to verify protocol specifications or small pieces of code. However, this technology is not usable by regular developers, and no general concept of bug-finding exists for security properties beyond safety, i.e., *hyperproperties* [10], such as non-interference [16].

In this paper we take the position that we should fill this gap in the arsenal of verification technology. We introduce the concept of *hypertests* (§2), which either constitute counter-examples to hyperproperties or serve as positive witnesses, providing some level of confidence in them. By systematically generating hypertests—*Automated hypertesting*—next-generation bug finding systems then will allow developers to automatically find violations of security properties, analogously to current testing systems for memory safety and assertions. We also briefly outline two possible applications of hypertesting in finding violations of secure information flow and cryptographic properties (§3).

2 Hypertesting

We will now recall the definition of hyperproperties (§2.1), introduce definitions for hypertests (§2.2) and k-hypertests (§2.3), and propose ideas for automating their generation (§2.4).

2.1 Hyperproperties

The classic types of properties used in verification are safety and liveness. An assertion in a program or the absence of buffer overflows are examples of safety properties; a typical liveness property is that an acquired resource is eventually freed. Safety and liveness properties can be categorized as *trace properties*, i.e., properties of individual execution traces of a program [10]. Any trace property can be formally defined as a set P of "legal" execution traces. A system then satisfies the property if the system's set T of possible executions is a subset of it ($T \subseteq P$).

However, many interesting security properties such as non-interference [16] or observational determinism [22] cannot be expressed as trace properties since they make statements about relations of multiple traces. Clarkson and Schneider [10] therefore introduced the notion of *hyperproperties*; a hyperproperty is a *set of sets of traces*, such that a system satisfies the hyperproperty H if the system's possible executions are an element of the hyperproperty ($T \in H$).

In principle, this makes hyperproperties considerably harder to prove or disprove, since all traces of a system may have to be reasoned about at the same time. Fortunately, many properties lie in subclasses that require reasoning only about k traces (k-safety). Observational determinism, for example, is a 2-safety hyperproperty that relates only two executions to each other [25]. This reduces the problem to verification of a safety property on a *self-composition* of the program. Barthe et al. [2] suggested directly combining the program with a renamed version of itself and proving that low-clearance values remain identical regardless of high-clearance values. Kovács et al. [19] apply self-composition on the control flow graph and use abstract interpretation for verification. In the context of testing, k-safety hyperproperties are particularly attractive, since k traces are sufficient to form a counterexample.

2.2 Hypertests

Hypertests relate to hyperproperties as regular test cases relate to trace properties. A test describes a subset of the program behavior and can be decided to either satisfy or violate the target property. For trace properties, a test is therefore just a single test input inducing a single trace of the program. To give a counter-example to liveness properties, this trace can also be infinite. Deciding whether a single trace of a program or system satisfies or violates a property is easier than constructing a proof for the entire program or system. This allows testing to be done more cheaply than correctness proofs, and often with full automation.

Moving to hyperproperties, a test inducing a single trace no longer suffices. Showing that a hyperproperty H is violated requires a set of program traces C such that there is no $T \in H$ with $C \subseteq T$. We therefore define a hypertest to be a (possibly infinite) set of test cases that induces a corresponding set of traces.

Note that by this definition, a trivial hypertest could be the set of all inputs, which would induce all traces and thus be equivalent to the complete concrete semantics of the program. This kind of hypertest would be clearly intractable, and in practical scenarios one will clearly aim for hypertests that execute in reasonable (and certainly finite) time.

Note further that deciding whether the hypertest satisfies or violates the hyperproperty is generally not as easy as it is for tests of trace properties. Consider again a hypertest encompassing the set of all inputs as a pathological example. Deciding whether this test satisfies or violates the hyperproperty is equivalent to verifying the full program. Thus we need to make sure not only that hypertests execute in finite time, but also that deciding the property on the induced set of traces is conceptually easy and can be fully automated. This will pose a considerable challenge for general hypertesting.

2.3 k-Hypertests

Fortunately many interesting properties do not require reasoning about infinite sets of traces. In the case of k-safety, k traces are sufficient to refute the property. Therefore, we can define corresponding k-hypertests that consist of exactly k traces. A failing k-hypertest constitutes a proof that the program does not satisfy the target property; a passing k-hypertest provides some degree of confidence that the property holds.

Just as with regular tests, it will be difficult to quantify the degree of confidence gained. An interesting avenue of work will be to develop practical coverage criteria for hypertests. On the one hand, simple metrics such as line coverage will have even less meaning for hypertests than they have for regular test suites. On the other hand, the total number of traces is unknown or infinite, so metrics like trace or path coverage cannot generally be meaningfully computed.

2.4 Automated Hypertesting

While developers may write hypertests themselves, as sets of unit tests, automatic generation of hypertests will be especially desirable.

Automated test case generation and bug finding using symbolic execution has received significant attention in recent years. The availability of powerful automated constraint solvers (SMT solvers) and the DART [15] and EXE [9] algorithms for combining symbolic and concrete execution have allowed its application to real world software [8]. Symbolic execution has so far focused mostly on finding memory safety violations or assertion failures in C code or binaries. Work on finding code injection vulnerabilities in JavaScript [24] has shown that symbolic execution can also be applied to higher-level programming languages and properties, but such use remains the exception.

Symbolic execution enumerates test inputs corresponding to the feasible execution paths of an application. It is thus ideally suited for generating hypertests. The main challenge will lie in guiding the path enumeration such that, where a program violates a property, the violating set of traces is found quickly. For regular trace properties, search strategies provide this kind of guidance for the path exploration, by trying to maximize various coverage criteria.

For k-safety properties, a test generation system can be built on the existing concept of self-composition [2]. A symbolic execution tool generating a single test case for a self-composed program will effectively generate two inputs for the same program, which constitutes a valid 2-hypertest.

3 Applications

We foresee two immediate applications for automatic hypertest generation in finding violations of secure information flow (§3.1) and of cryptographic properties (§3.2).

3.1 Testing Information Flow Properties

In the context of information flow, testing is related to dynamic policy enforcement and monitors. A security monitor runs alongside a program and terminates it once it can

no longer guarantee a security property. For noninterference, monitors can be proven to be equivalent to type systems, disregarding termination [23]. Monitors (like type systems) have to be conservative and are thus prone to false positives; this makes them unsuitable for a direct application in testing when the goal is to generate guaranteed counterexamples.

Based on the observation that many interesting information flow properties are in fact 2-safety properties, one may develop suitable efficient algorithms for directed testing that can produce corresponding tests and counterexamples. A valid counterexample to observational determinism [22], for example, could consist of two traces that yield different low-clearance outputs on identical low-clearance inputs (because the low-clearance outputs depend on high-clearance inputs). This idea is embodied in existing work on random-testing information flow properties [17], but no systematic approach exists. Self-composition [2, 19] is not generally necessary for testing, but can quickly yield a test generation engine with an existing symbolic execution tool (see §2.4). A key question will be to find suitable novel search strategies that can efficiently guide test generation and symbolic execution towards pairs of paths that violate the property.

3.2 Cryptographic Bugs

Cryptographic properties, such as the distribution of random numbers or the size of the key space, are an important example of properties that go beyond basic safety. Such a property is easily violated, e.g., by a bad typecast truncating a random number. Unlike bugs that affect observable application behavior, however, these "cryptobugs" are difficult to spot. The introduced cryptographic weakness is subtle and can persist for long as it does not affect the typical user experience.

A number of such weaknesses became more widely known: (i) the "Cryptocat" chat client was discovered to be insecure because a string of decimals from a random number generator had been mistaken for an array of bytes, reducing the possible key space [26]; (ii) the "Bitmessage" peer-to-peer messenger was found, among other problems, to lack a secure block chaining mode [20]; (iii) the "enigmail" e-mail encryption plugin could sometimes send an e-mail completely unencrypted to BCC recipients [11]; (iv) finally, over 40% of Android applications using cryptography were found to have hard-coded cryptographic keys [27].

Not all "cryptobugs" in application software manifest as violations of hyperproperties; some are much simpler to find and stem from violations of API contracts and simple numeric properties like "parameter x must not be constant". Existing light-weight static analysis and test-generation methods for safety properties can be adopted to find these kinds of bugs.

Hyperproperties come into play when moving on to properties of distributions such as "parameter y must be uniformly distributed". For instance, known PRNGs can be assumed correct and annotated to output uniformly distributed values of a specified range. The application code may then transform this distribution before feeding it into other annotated cryptographic functions. These are no longer regular safety properties; a single constant flowing into a key parameter is only a bug if that path has a probability greater than 1 divided by the size of the key space.

4 Related Work

Testing has previously attracted the attention of the information flow community. Birgisson et al. [6] showed how to use testing to insert security upgrade annotations automatically whenever a monitor reports a violation. Hriţcu et al. [17] proposed to test dynamic (e.g., monitors) or static (e.g., type systems) information flow enforcement mechanisms using the QuickCheck random testing system. Their strategy is to randomly generate and run programs conforming to the mechanism and then to independently verify the desired security property. Since their focus is on verifying the mechanism, the generated programs consist of only a few instructions. Interestingly, Hriţcu et al. note that using symbolic execution instead of random testing appears promising but found the existing engine they tried to be ineffective.

In recent work, Phan et al. [21] successfully used symbolic execution with self-composition for quantifying information flow. They exhaustively enumerate paths and compute the number of outputs by counting models for path conditions (following [14]) of leaking paths. Their success on small benchmarks (up to 40 lines of code) provides initial evidence that symbolic execution is viable for analyzing security properties.

Traditional verification of cryptographic protocols is based on symbolic analysis of high-level Dolev-Yao models [12]. Verification of cryptographic implementations is a more recent endeavor; particularly successful have been type systems such as those in the F7 verification system [4], which led the way to later work on verifying cryptographic implementations using refinement types [1, 5, 3]. While type systems are applicable to real code, they require careful annotations and are intrinsically prone to false positives.

5 Conclusion

Today's users rely on the security of a diverse range of application code, much of it written by application developers who are not security experts. Automated generation of hypertests promises to yield a system for identifying a wide range of security violations fully automatically, without requiring expert users. Since no complete proofs with manual intervention are required, such a system would allow even developers of commodity software to easily find security bugs in their code.

If fully developed and broadly deployed, this has the chance to increase the overall robustness and security of the IT ecosystem and help fight the all too common sentiment that "the bad guys are winning".

Acknowledgments

The author would like to thank David Cock, Sergio Maffeis, and José Santos for valuable discussions that helped to shape the ideas in this paper. This work has been partially supported by EPSRC grant EP/L022710/1.

References

[1] M. Backes, M. Maffei, and D. Unruh. "Computationally sound verification of source code". In: *Proc. 17th ACM SIGSAC Conf. Computer and Communications Security (CCS 2010)*. ACM, 2010, pp. 387–398.

[2] G. Barthe, P. R. D'Argenio, and T. Rezk. "Secure Information Flow by Self-Composition". In: *17th IEEE Computer Security Foundations Workshop (CSFW 2004)*. IEEE Computer Society, 2004, pp. 100–114.

[3] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin. "Probabilistic relational verification for cryptographic implementations". In: *Proc. 41st Annu. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2014)*. 2014, pp. 193–206.

[4] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. "Refinement Types for Secure Implementations". In: *Proc. 21st IEEE Computer Security Foundations Symp. (CSF 2008)*. IEEE Computer Society, 2008, pp. 17–32.

[5] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. "Implementing TLS with Verified Cryptographic Security". In: *IEEE Symp. Security and Privacy (S&P 2013)*. 2013, pp. 445–459.

[6] A. Birgisson, D. Hedin, and A. Sabelfeld. "Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing". In: *Proc. 17th European Symp. Research in Computer Security (ESORICS 2012)*. 2012, pp. 55–72.

[7] D. Bongard. *Offline bruteforce attack on WiFi Protected Setup*. Presentation at Passwordscon 2014, Las Vegas. 2014.

[8] C. Cadar, D. Dunbar, and D. R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proc. 8th Symp. Operating Systems Design and Implementation (OSDI 2008)*. 2008, pp. 209–224.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. "EXE: automatically generating inputs of death". In: *Proc. 13th ACM SIGSAC Conf. Computer and Communications Security (CCS 2006)*. ACM, 2006, pp. 322–335.

[10] M. R. Clarkson and F. B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[11] L. Constantin. *Encryption failures fixed in popular PGP email security tool Enigmail*. PCWorld. 2014. URL: `http://www.pcworld.com/article/2604880/encryption-failures-fixed-in-popular-pgp-email-security-tool-enigmail.html`.

[12] D. Dolev and A. C.-C. Yao. "On the security of public key protocols". In: *IEEE Trans. Information Theory* 29.2 (1983), pp. 198–207.

[13] P. Ducklin. *Anatomy of a "goto fail" – Apple's SSL bug explained, plus an unofficial patch for OS X!* Sophos – Naked Security. Feb. 2014. URL: `https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/`.

[14] A. Filieri, C. S. Pasareanu, and W. Visser. "Reliability analysis in symbolic pathfinder". In: *35th Int. Conf. Software Engineering (ICSE 2013)*. 2013, pp. 622–631.

[15] P. Godefroid, N. Klarlund, and K. Sen. "DART: directed automated random testing". In: *Proc. ACM SIGPLAN 2005 Conf. Programming Language Design and Implementation (PLDI 2005)*. ACM, 2005, pp. 213–223.

[16] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *IEEE Symp. Security and Privacy (S&P 1982)*. 1982, pp. 11–20.

[17] C. Hritcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. A. de Amorim, and L. Lampropoulos. “Testing noninterference, quickly”. In: *ACM SIGPLAN Int. Conf. Functional Programming (ICFP'13)*. 2013, pp. 455–468.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. “seL4: formal verification of an OS kernel”. In: *Proc. 22nd ACM Symp. Operating Systems Principles (SOSP 2009)*. ACM, 2009, pp. 207–220.

[19] M. Kovács, H. Seidl, and B. Finkbeiner. “Relational abstract interpretation for the verification of 2-hypersafety properties”. In: *Proc. 20th ACM SIGSAC Conf. Computer and Communications Security (CCS 2013)*. ACM, 2013, pp. 211–222.

[20] S. D. Lerner. *Bitmessage v1.0: completely broken crypto*. 2012. URL: `http://bitslog.wordpress.com/2012/11/30/bitmessage-completely-broken-crypto/`.

[21] Q. Phan, P. Malacaria, C. S. Pasareanu, and M. d'Amorim. “Quantifying information leaks using reliability analysis”. In: *Proc. Int. Symp. Model Checking of Software (SPIN 2014)*. 2014, pp. 105–108.

[22] A. W. Roscoe. “CSP and determinism in security modelling.” In: *IEEE Symp. Security and Privacy (S&P 1995)*. IEEE Computer Society, 1995, pp. 114–127.

[23] A. Sabelfeld and A. Russo. “From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research”. In: *7th Int. Andrei Ershov Memorial Conf. Perspectives of Systems Informatics (PSI 2009)*. 2009, pp. 352–365.

[24] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. “A Symbolic Execution Framework for JavaScript”. In: *IEEE Symp. Security and Privacy (S&P 2010)*. IEEE Computer Society, 2010, pp. 513–528.

[25] T. Terauchi and A. Aiken. “Secure Information Flow as a Safety Problem”. In: *12th Int. Symp. Static Analysis (SAS 2005)*. Vol. 3672. LNCS. Springer, 2005, pp. 352–367.

[26] S. Thomas. *DecryptoCat*. 2013. URL: `http://tobtu.com/decryptocat.php`.

[27] Veracode. *State of software security report: The intractable problem of insecure software*. Dec. 2011.